Exhibit A to

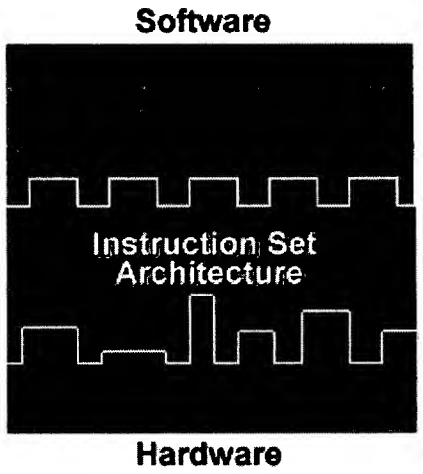
Response to Office Action

Web Page Discussing Microcode

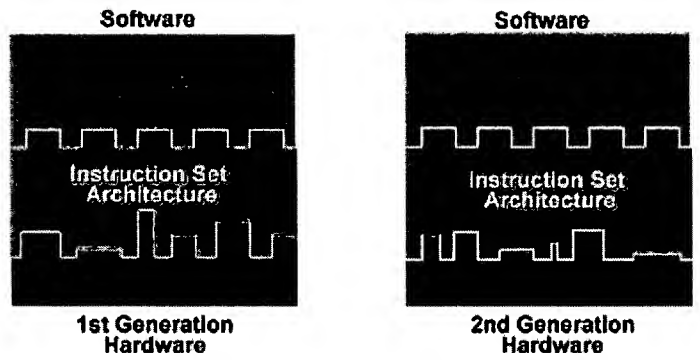
The Future of x86 and the Concept of the Instruction Set Architecture

by Hannibal

IBM solved this problem with the introduction of the Instruction Set Architecture (ISA) and the *microcode engine* in the 60's. Specifically, the launch of the microcoded IBM System/360 ushered in the era of modern computer architecture, where the information that the programmer needed to know to program the machine was abstracted from the actual implementation of that machine. Once the design and specification of the instruction set, or the set of instructions available to a programmer for writing programs, was separated from the nitty gritty details of a particular machine's design it meant that that programs written for a particular ISA could now run on any machine that implemented that ISA.



Thus the ISA provided a standardized way to expose the features of a system's hardware that allowed manufacturers to innovate and fine-tune that hardware for performance without worrying about breaking the existing software base. You could release a first generation product with a particular ISA, then work on speeding up the implementation of that same ISA for the second generation product, which would be backwards compatible with the first generation. We take all this for granted now, but before the IBM System/360, binary compatibility between different machines of different generations didn't exist.



The blue layer in the above diagram simply represents the ISA as an *abstract model* of a machine for which a programmer writes programs. As I mentioned before, the technical innovation that made this abstract layer possible was something called the *microcode engine*. A microcode engine is sort of like a CPU within a CPU. It consists of a microcode ROM that holds microcode programs and an execution unit that executes those programs. The job of each of these microcode programs is to translate a particular instruction into a series of signals that controls the innards of the chip. When a System/360 instruction is executed, the microcode ROM reads the instruction in, then loads and executes the proper microcode program that corresponds to that instruction; that program orchestrates the dance of memory accesses and functional unit activations that actually does the number crunching (or whatever else) that the instruction has commanded the machine to do.

By doing things this way, all programs are effectively running in emulation. So the ISA represents a sort of idealized model, emulated by the underlying hardware, on the basis of which you can design applications. This emulation means that between iterations of a product line you can change everything about the way the CPU executes a program, and all you have to do is rewrite the microcode program each time so that the programmer will never have to be aware of the hardware differences because the ISA hasn't changed a bit. Microcode engines still show up in modern CPUs. The Athlon uses one for the part of its decoding path that decodes the larger x86 instructions. (See my "Into the K7" for more details.)

So the key to understanding the above diagram is that the blue layer represents a layer of abstraction that hides the complexity of the underlying hardware from the programmer. The blue layer is not a

hardware layer (that's the gray one) and it's not a software layer (that's the red one) but it's a *conceptual* layer. Think of it almost like a user interface that hides the complexity of a machine from the user. All the user needs to know to use the machine is how to close windows, launch programs, find files, etc. The UI (and by this I mean the conceptual paradigm--windows, icons, etc.--and not the software that implements the UI) exposes the machine's power and functionality to the user in a way that he or she can understand and use. And whether that UI appears on a PDA or on a desktop machine, the user still knows how to use it to command the machine.

The main drawback to using microcode to implement an ISA is that the microcode engine was, in the beginning, slower than direct execution. (Modern microcode engines are about 99% as fast as direct execution.) However, the ability to separate ISA design from microarchitectural implementation was so significant for the development of modern computing that the small speed hit incurred was well worth it.

The advent of the RISC movement in the 70's saw a couple of changes to the above scheme. First and foremost, RISC was all about throwing stuff overboard in the name of speed. So the first thing to go was the microcode engine. Microcode had allowed ISA designers to go butt nutty with instruction sets, adding in all sorts of crazy instructions that programmers rarely used. More instructions meant more microcode ROM, larger die sizes, more power, etc. Since RISC was more about less, the microcode engine got the axe. RISC reduced the number of instructions in the instruction set and reduced the sizes of each individual instruction so that this smaller, faster and more lightweight instruction set could be more easily implemented directly in hardware, without a bulky microcode engine.

While RISC designs went back to the old method of direct execution of instructions, they kept the concept of the ISA intact. Computer architects had by this time learned the immense value of not breaking backwards compatibility with old software, and they weren't about to go back to the bad old days of wedding software to a single product. The ISA was still the optimal solution for the problem of easily and consistently exposing hardware functionality to programmers so that software could be used across a range of machines.

Next: Today's ISA

On this page:
Technology

instruction set

Technology

Computer Desktop Encyclopedia

instruction set

The repertoire of machine language instructions that a computer can follow (from a handful to several hundred). It is a major architectural component and is either built into the CPU or into microcode. Instructions are generally from one to four bytes long.

Wikipedia

instruction set

An **instruction set**, or **instruction set architecture** (ISA), describes the aspects of a computer architecture visible to a programmer, including the native datatypes, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O (if any).

An ISA is a specification of the set of all binary codes (opcodes) that are the native form of commands implemented by a particular CPU design. The set of opcodes for a particular ISA is also known as the machine language for the ISA.

"Instruction set architecture" is sometimes used to distinguish this set of characteristics from the microarchitecture, which is the set of processor design techniques used to implement the instruction set (including microcode, pipelining, cache systems, and so forth). Computers with different microarchitectures can share a common instruction set. For example, the Intel Pentium and the AMD Athlon implement nearly identical versions of the x86 instruction set, but have radically different internal designs. This concept can be extended to unique ISAs like TIMI present in the IBM System/38 and IBM IAS/400. TIMI is an ISA that is implemented as low-level software and functionally resembles what is now referred to as a virtual machine. It was designed to increase the longevity of the platform and applications written for it, allowing the entire platform to be moved to very different hardware without having to modify any software except that which comprises TIMI itself. This allowed IBM to move the AS/400 platform from an older CISC architecture to the newer POWER architecture without having to rewrite any parts of the OS or software associated with it.

When designing microarchitectures, engineers use Register Transfer Language (RTL) to define the operation of each instruction of an ISA.

An ISA can also be emulated in software by a interpreter. Due to the additional translation needed for the emulation, this

is usually slower than directly running programs on the hardware implementing that ISA. Today, it is common practice for vendors of new ISAs or microarchitectures to make software emulators available to software developers before the hardware implementation is ready.

List of ISAs

This list is far from comprehensive as old architectures died out and new ones invented on a continual basis. There is also a plethora of commercially available microprocessors and microcontrollers implementing ISAs in all shapes and sizes. Customized ISAs are also quite common in some applications, e.g. ARC International, application-specific integrated circuit, FPGA, and reconfigurable computing. Also see history of computing hardware.

ISAs commonly implemented in hardware

- Alpha AXP (DEC Alpha)
- ARM (Acorn RISC Machine) (Advanced RISC Machine now ARM Ltd)
- IA-64 (Itanium)
- MIPS
- Motorola 68k
- PA-RISC (HP Precision Architecture)
- IBM POWER
- PowerPC
- SPARC
- SuperH
- System/360
- Tricore (Infineon)
- Transputer (STMicroelectronics)
- VAX (Digital Equipment Corporation)
- x86 (IA-32, Pentium, Athlon) (AMD64, EM64T)

ISAs commonly implemented in software with hardware incarnations

- p-Code (UCSD p-System Version III on Western Digital Pascal Micro-Engine)
- Java virtual machine (ARM Jazelle, PicoJava)
- FORTH

ISAs never implemented in hardware

- SECD machine
- ALGOL Object Code

See also

Categories of ISA

- application-specific integrated circuit (ASIC) fully custom ISA
- CISC
- digital signal processor
- graphics processing unit
- reconfigurable computing
- RISC
- vector processor
- VLIW

Examples of commercially available ISA

- central processing unit
- microcontroller
- microprocessor

Others

- computer architecture
- CPU design
- emulator
- hardware abstraction layer
- Register Transfer Language
- virtual machine
- Atmel AVR instruction set
- Streaming SIMD Extensions (SEE) instruction set
- SSE2 IA-32 SIMD instruction set

External links

- Mark Smotherman's Historical Computer Designs Page (*http://www.cs.clemson.edu/~mark/hist.html*)

Mentioned In

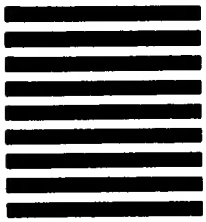
instruction set is mentioned in the following topics:

command set (technology)	instruction repertoire (technology)
CISC (abbreviation)	RISC (abbreviation)
Microarchitecture	x64
MSPA	microcode (technology)
Instruction Set Architecture	Visual Instruction Set
	More>

Exhibit B to

Response to Office Action

Excerpts from Intel Architecture User's Manual



intel®

Intel Architecture Software Developer's Manual

Volume 2: Instruction Set Reference

NOTE: The *Intel Architecture Software Developer's Manual* consists of three volumes: *Basic Architecture*, Order Number 243190; *Instruction Set Reference*, Order Number 243191; and the *System Programming Guide*, Order Number 243192. Please refer to all three volumes when evaluating your design needs.

1997

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

Opcode	Instruction	Description
F3 6C	REP INS *r/m8*, DX	Input (E)CX bytes from port DX into ES:[(E)DI]
F3 6D	REP INS *r/m16*,DX	Input (E)CX words from port DX into ES:[(E)DI]
F3 6D	REP INS *r/m32*,DX	Input (E)CX doublewords from port DX into ES:[(E)DI]
F3 A4	REP MOVS *m8,m8*	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS *m16,m16*	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS *m32,m32*	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI]
F3 6E	REP OUTS DX,*r/m8*	Output (E)CX bytes from DS:[(E)SI] to port DX
F3 6F	REP OUTS DX,*r/m16*	Output (E)CX words from DS:[(E)SI] to port DX
F3 6F	REP OUTS DX,*r/m32*	Output (E)CX doublewords from DS:[(E)SI] to port DX
F3 AC	REP LODS AL	Load (E)CX bytes from DS:[(E)SI] to AL
F3 AD	REP LODS AX	Load (E)CX words from DS:[(E)SI] to AX
F3 AD	REP LODS EAX	Load (E)CX doublewords from DS:[(E)SI] to EAX
F3 AA	REP STOS *m8*	Fill (E)CX bytes at ES:[(E)DI] with AL
F3 AB	REP STOS *m16*	Fill (E)CX words at ES:[(E)DI] with AX
F3 AB	REP STOS *m32*	Fill (E)CX doublewords at ES:[(E)DI] with EAX
F3 A6	REPE CMPS *m8,m8*	Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI]
F3 A7	REPE CMPS *m16,m16*	Find nonmatching words in ES:[(E)DI] and DS:[(E)SI]
F3 A7	REPE CMPS *m32,m32*	Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI]
F3 AE	REPE SCAS *m8*	Find non-AL byte starting at ES:[(E)DI]
F3 AF	REPE SCAS *m16*	Find non-AX word starting at ES:[(E)DI]
F3 AF	REPE SCAS *m32*	Find non-EAX doubleword starting at ES:[(E)DI]
F2 A6	REPNE CMPS *m8,m8*	Find matching bytes in ES:[(E)DI] and DS:[(E)SI]
F2 A7	REPNE CMPS *m16,m16*	Find matching words in ES:[(E)DI] and DS:[(E)SI]
F2 A7	REPNE CMPS *m32,m32*	Find matching doublewords in ES:[(E)DI] and DS:[(E)SI]
F2 AE	REPNE SCAS *m8*	Find AL, starting at ES:[(E)DI]
F2 AF	REPNE SCAS *m16*	Find AX, starting at ES:[(E)DI]
F2 AF	REPNE SCAS *m32*	Find EAX, starting at ES:[(E)DI]

Description

Repeats a string instruction the number of times specified in the count register ((E)CX) or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The behavior of the REP prefix is undefined when used with non-string instructions.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct.

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix (Continued)

All of these repeat prefixes cause the associated instruction to be repeated until the count in register (E)CX is decremented to 0 (see the following table). (If the current address-size attribute is 32, register ECX is used as a counter, and if the address-size attribute is 16, the CX register is used.) The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the (E)CX register with a JECXZ instruction or by testing the ZF flag with a JZ, JNZ, and JNE instruction.

Repeat Conditions

Repeat Prefix	Termination Condition 1	Termination Condition 2
REP	ECX=0	None
REPE/REPZ	ECX=0	ZF=0
REPNE/REPNZ	ECX=0	ZF=1

When the REPE/REPZ and REPNE/REPNZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPNE, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute.

A REP STOS instruction is the fastest way to initialize a large block of memory.

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix (Continued)

Operation

```
IF AddressSize = 16
    THEN
        use CX for CountReg;
    ELSE (* AddressSize = 32 *)
        use ECX for CountReg;
FI;
WHILE CountReg ≠ 0
    DO
        service pending interrupts (if any);
        execute associated string instruction;
        CountReg ← CountReg – 1;
        IF CountReg = 0
            THEN exit WHILE loop
        FI;
        IF (repeat prefix is REPZ or REPE) AND (ZF=0)
        OR (repeat prefix is REPNZ or REPNE) AND (ZF=1)
            THEN exit WHILE loop
        FI;
    OD;
```

Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

Exceptions (All Operating Modes)

None; however, exceptions can be generated by the instruction a repeat prefix is associated with.

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☒ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☒ FADED TEXT OR DRAWING
- ☐ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☐ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☐ GRAY SCALE DOCUMENTS
- ☐ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: _____

IMAGES ARE BEST AVAILABLE COPY.
As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.